

APPENDIX G


```

{
t_index j, k;

num_instances = 0;
num_states = num_states_of_symbol;
num_mixes = num_mixes_of_symbol;

tran.Destroy_And_ReDim(num_states, num_states);
occ.Destroy_And_ReDim(num_states);
mu.Destroy_And_ReDim(num_states);

for(j=0; j<num_states; j++)
{
mu[j].Destroy_And_ReDim(num_mixes);

for(k=0; k<num_mixes; k++)
mu[j][k].Destroy_And_ReDim(obs_size);
}

if(full_covs)
{
full_cov.Destroy_And_ReDim(num_states);

for(j=0; j<num_states; j++)
{
full_cov[j].Destroy_And_ReDim(num_mixes);
for(k=0; k<num_mixes; k++)
{
full_cov[j][k].Destroy_And_ReDim(obs_size,
obs_size);
}
}
}
else{
diag_va.Destroy_And_ReDim(num_states);
for(j=0; j<num_states; j++)
{
diag_va[j].Destroy_And_ReDim(num_mixes);
for(k=0; k<num_mixes; k++)
diag_va[j][k].Destroy_And_ReDim(obs_size);
}
}

c.Destroy_And_ReDim(num_states, num_mixes);

return;

```

```

    }

void StatisticsAccumulators::Reset_Parameters()
{
    t_index j, k;
    num_instances = 0;

    tran.Set(0.0);
    occ.Set(0.0);

    for(j=0; j<num_states; j++)
        for(k=0; k<num_mixes; k++)
            mu[j][k].Set(0.0);

    if(full_cov.Dim()>0)
        for(j=0; j<num_states; j++)
            for(k=0; k<num_mixes; k++)
                full_cov[j][k].Set(0.0);
    else for(j=0; j<num_states; j++)
        for(k=0; k<num_mixes; k++)
            diag_va[j][k].Set(0.0);

    c.Set(0.0);
    return;
}

// *****
//
//                      REESTIMATE      MODELS PARAMETERS
//
//                      UNTIL THE DESIRED CONVERGENCE VALUE IS REACHED
//
// *****

Boolean ModelsSimultaneousTraining::Symbol_Models_Optimisation()
{
    t_real average_likelyhood;
    t_index iteration_counter = 0, i;

    ofstream file;
    file.open("likelyhood.int");
    file.close();

    if(load_accs)
        Update_Models();

```

```

else{
    mstat<<"Start parameter reestimation cycle.";

    do        {
        for(i=0;i<HMM_accs.Dim();i++)
            HMM_accs[i].Reset_Parameters();
        ReEstimate_Parameters(average_likelyhood, dbase);
        Update_Models();
        iteration_counter++;

        file.open("likelyhood.int",ios::app);
        file<<"Average likelyhood at iteration
"<<iteration_counter
            <<":"<<average_likelyhood<<"\n";
        file.close();

        Store_Iteration_Model(iteration_counter);
        dbase.Restart();

        } while(iteration_counter<max_num_iteration);

    mstat<<"Training set likelyhood: "<<average_likelyhood;
    }

    file.close();

    if(store_accs)
        Store_Statistic_Accs(accs_file);
    return (Boolean)TRUE;
}

void ModelsSimultaneousTraining::Store_HMM_Models()
{
    t_index symb;
    String f_name;

    for(symb=0; symb<HMM_defs.Dim(); symb++)
        HMM_defs[symb].Write(models_file_output);

    return;
};

void ModelsSimultaneousTraining::Store_Iteration_Model(t_index iteration)
{
    t_index symb;
    ofstream file;

```

```

String f_name;

f_name<<"iteration_"<<iteration<<".emb";
    file.open(f_name);

    Write_Header_Of_File_Model(f_name,      dbase.Snd_Type(),
                                dbase.Label_Type(), dbase.Db_File_List_Name
                                ( ),dbase.Window_Lenght(),
                                dbase.Window_Overlap(), 39, FALSE);

    file.close();
    for(symb=0; symb<HMM_defs.Dim(); symb++)
        {
            HMM_defs[symb].Write(f_name);
        }

    return;
}

// Init_Emb_Train: initialise dbase, training structures and load models.
void ModelsSimultaneousTraining::Configure(const String& config_file)
{
    ConfigFile conf;
    t_index symb_num, position;

    features.Configure(config_file);
    dbase.Configure(config_file, TRUE);
    symb_num=dbase.Get_Num_Of_Symbols();
    conf.Open_File(config_file);
    conf.Get_String_Opt("ModelsSimultaneousTraining", "modelsfilename",
models_file_input);

    max_num_iteration = conf.Get_Unsigned_Opt
("ModelsSimultaneousTraining", "MaxNumIteration");
    reest_means=conf.Get_Boolean_Opt("ModelsSimultaneousTraining",
"ReestimateMeans");
    reest_variances = conf.Get_Boolean_Opt
("ModelsSimultaneousTraining", "ReestimateVariances");

    reest_weights   = conf.Get_Boolean_Opt
("ModelsSimultaneousTraining", "ReestimateWeights");
    reest_transitions = conf.Get_Boolean_Opt
("ModelsSimultaneousTraining", "ReestimateTransitions");
    min_instance_number = conf.Get_Unsigned_Opt

```

```

("ModelsSimultaneousTraining", "MinInstanceNumber");
    pruning_threshold = conf.Get_Real_Opt("ModelsSimultaneousTraining",
"PruningThreshold");
    min_var_value = conf.Get_Real_Opt("ModelsSimultaneousTraining",
"MinimunVarianceValueIfDiagonal");
    const_to_add_min_var= conf.Get_Real_Opt
("ModelsSimultaneousTraining", "ConstToAddMinVar");

    load_accs = conf.Get_Boolean_Opt("ModelsSimultaneousTraining",
"LoadAccumulators");
    if(load_accs)
        conf.Get_String_Opt("ModelsSimultaneousTraining",
"AccsList", list_file);

    store_accs = conf.Get_Boolean_Opt("ModelsSimultaneousTraining",
"StoreAccumulators");
    if(store_accs)
        conf.Get_String_Opt("ModelsSimultaneousTraining",
"AccsFile", accs_file);

    models_file_input.Is_SubString_Inside(".", position);
    models_file_output.Destroy_And_ReDim(position);
    models_file_output.Take(models_file_input,0,position);
    models_file_output<<"emb";

    HMM_defs.Destroy_And_ReDim(symb_num);
    Load_Models_Parameters();
    if(load_accs)
    {
        ifstream file_list;
        String file_name;

        file_list.open(list_file, ios::in|ios::nocreate);
        if(file_list.fail())
            merr<<"Cannot open file of list of statistics
accumulators";

        while(!file_list.eof())
        {
            file_list>>file_name;
            if(file_list.eof() AND file_name[0]==EOF)
                return;
            Load_Statistic_Accs(file_name);
        }
    }

```

```

return;
};

```

```

// *****
//
// REESTIMATE PARAMETRS
*
//
// *****

```

```

t_real ModelsSimultaneousTraining::Sequences_Total_LProb(const VetDouble&
sequences_lprob)

```

```

{
    t_real temp;
    t_index i;

    temp = sequences_lprob[0];
    for(i=1;i<sequences_lprob.Dim();i++)
        temp+=sequences_lprob[i];

    return temp;
};

```

```

void ModelsSimultaneousTraining::ReEstimate_Parameters(t_real&
training_set_likelyhood,

```

```

                                DbaseVoc& dbase)
{
    Boolean not_end_of_dbase, is_new_file;

    t_index i, file_counter;
    VetDouble vetsmp;
    t_string_list label_list;
    t_real sequence_lprob;
    VetDouble sequences_lprob;
    static VetDouble observation_vet;
    VetDoubleList pred_list;
    T = 0;

    not_end_of_dbase = (Boolean)TRUE;
    pred_list.Destroy_And_ReDim(features.Max_Delta_Feature_Order());

    while(not_end_of_dbase)
        {

```

```

// indicating in "is_new_file" if it is the first frame of
a
// new sentence and the condition "not_end_of_dbase"
not_end_of_dbase=dbase.Get_Sequential_Vet(vetsmp,
is_new_file);

if( (T!=0) AND (is_new_file==TRUE OR
not_end_of_dbase==FALSE ) )
// i.e. if an file has been read
{
// TRAINING STEP: apply forward-backward procedure
// to given file and update parameter
accumulators

Assert(T == file.Dim());
sequence_lprob = Update_Parameters_With_New_file(file,

label_idxes_list);
sequences_lprob.Append(sequence_lprob);
file.Reset();
cout<<"file_num: "<<sequences_lprob.Dim()<<'\\t';
cout<<"num frames: "<<T<<endl;
T=0;
}

if ( not_end_of_dbase )
{
if(is_new_file)
{
dbase.Get_Label_List_Of_Actual_file
(label_idxes_list);

Q = label_idxes_list.Dim();
for(i=0;i<pred_list.Dim();i++)
// just a little trick: starting
frames initialized with
// the value of the first frame: in
order to have a starting value

pred_list[i] = vetsmp;
features.Get_Previous_Frames_Info
(pred_list, dbase.Smp_Rate());
}

features.Extract(observation_vet, vetsmp,
dbase.Smp_Rate());

// memorize in file the samples of the next file
// of training data

```

```

        file.Append(observation_vet);
        T++;
    }

    } // end while

    file_counter = sequences_lprob.Dim();
    training_set_likelihoood = Sequences_Total_LProb(sequences_lprob)/
(t_real)file_counter;

    return;
};

t_real ModelsSimultaneousTraining::Update_Parameters_With_New_file
(const Bi_D_List& file, const VetULong&
label_idxes_list)
{
    t_index q, t, startq, endq;
    t_real file_lprob;

    VetDouble obs; // observation vector
    VetULong top_label; // top of pruning beam
    VetULong bottom_label; // bottom of pruning beam

    file_lprob = Compute_Beta(top_label, bottom_label, file,
label_idxes_list);

    if (file_lprob > LOGZERO) // if models fit current file
    {
        //alfa computation and parameters updating
        //performed only if model fits actual file
        Init_Alpha(label_idxes_list);

        //update occurrence counters of the HMMs
        for (q=0;q<Q;q++)
            HMM_accs[label_idxes_list[q]].num_instances++;
        //for all frames
        for (t=0;t<T;t++)
        {
            Get_Observation(obs, file, t);
            if (t>0)
                Compute_Alpha_At_Time_t(startq, endq,
file_lprob, top_label,

```

```

        bottom_label, t, label_idxes_list);
        else startq = endq = 0;
        // update parameters only on a significant sequence
        // ( starting at label no. startq and ending at
label endq)
        // of labels inside the actual file
        for (q=startq;q<=endq;q++)
        {
            Update_Occourrence_Counter(HMM_accs
[label_idxes_list[q]], t,q, file_lprob);

            if (reest_means OR reest_variances OR
reest_weights)
                // HMM_defs[label_list[q]] is the
hmm
                // of the q-th labelof the file
                Up_Mix_Parms(HMM_defs
[label_idxes_list[q]], HMM_accs[label_idxes_list[q]],
q, t, obs,
file_lprob);

            if (reest_transitions)
                Up_Tran_Parms(HMM_defs
[label_idxes_list[q]], HMM_accs[label_idxes_list[q]],
file_lprob, q, Q,
t, T);
        } // endfor q
    } // endfor t
} // end if

return file_lprob;
}

```

```

// InitPruneStats: initialise pruning stats
void ModelsSimultaneousTraining::Init_Prune_Stats()
{
    maxBeamWidth = 0;
    maxAlphaBeta = LOGZERO;
    minAlphaBeta = 1.0;
}

```

```

// CheckPruning: record peak alfa.beta product and position
void ModelsSimultaneousTraining::CheckPruning(t_index t, const t_index

```

```

beam_top, const t_index beam_bottom,

    EmbCodebook& act_hmm)
{
    t_signed margin;
    t_index i, q, bestq, besti, Nq;
    t_real l, maxL;

    bestq = besti = 0;
    maxL = LOGZERO;
    for (q=beam_bottom ; q<=beam_top ; q++)
    {
        Nq = act_hmm.num_states;
        for (i=1;i<Nq-1;i++)
        {
            if((l=act_alfa[q][i] + beta[q][t][i]) > maxL)
            {
                bestq = q;
                besti = i;
                maxL=l;
            }
        }
    }
    if (maxL > maxAlphaBeta)
        maxAlphaBeta = maxL;
    if (maxL < minAlphaBeta)
        minAlphaBeta = maxL;
    margin = beam_top-beam_bottom+1;
    if (margin>maxBeamWidth)
        maxBeamWidth = margin;
}

// CreateInsts: create array of hmm instances indexes
void ModelsSimultaneousTraining::Create_Insts(VetULong& label_idxes_list,

const t_string_list& label_list, DbaseVoc& dbase)
{
    t_index q;

    label_idxes_list.Destroy_And_ReDim(label_list.Dim());
    for (q=0;q<label_list.Dim();q++)
        label_idxes_list[q] = dbase.Translate_Symbol(label_list
[q]);

    return;
}

```

```

// SetBeta: allocate and calculate beta and oprob matrices
t_real ModelsSimultaneousTraining::Compute_Beta(VetULong& qHi, VetULong&
qLo,
const Bi_D_List&
whole_file, const VetULong& label_idxes_list)
{
    t_index t,q;
    t_index i, j, Nq, q_at_gMax, startq, endq, last_q;
    VetDouble bqt,bqt1,outprob,maxP;
    t_real x, a, y, gMax, lMax;
    EmbCodebook *act_HMM;
    t_real pr;

    // Create Storage Space - excluding actual data arrays

    // storage for min and max q values
    qHi.Destroy_And_ReDim(T);
    qLo.Destroy_And_ReDim(T);

    // dimensionate beta and obs_lprob
    beta.Destroy_And_ReDim(Q);
    obs_lprob.Destroy_And_ReDim(Q);
    for(q=0;q<Q;q++)
    {
        beta[q].Destroy_And_ReDim(T);
        obs_lprob[q].Destroy_And_ReDim(T);
    }
    maxP.Destroy_And_ReDim(Q); // for calculating
beam width

    act_HMM = &HMM_defs[label_idxes_list[Q-1]];
    Nq = act_HMM->num_states;
    beta[Q-1][T-1].Destroy_And_ReDim(Nq);
    beta[Q-1][T-1][Nq-1]=0.0;

    for (i=1;i<Nq-1;i++)
        beta[Q-1][T-1][i] = act_HMM->trans_mat[i][Nq-1];

    beta[Q-1][T-1][0]=LOGZERO;
    qHi[T-1] = qLo[T-1] = Q-1;
    Compute_Obs_LProbs(whole_file[T-1], T-1, qHi[T-1],
        qLo[T-1], label_idxes_list);
    Assert(T>=2);

```

```

for (t=T-2;t!=(t_index)(-1);t--)
{
    gMax = LOGZERO;    // max value of beta at time t
    if(t>=qHi[t+1]) startq=qHi[t+1];
    else startq = t;

    if (0==qLo[t+1]) endq = 0;
    else endq = qLo[t+1]-1;

    Assert(startq>=endq);
    for (q=startq;q!=(t_index)(endq-1);q--)
    {
        lMax = LOGZERO; // max value of beta in model q
        act_HMM = &HMM_defs[label_idxes_list[q]];
        Nq = act_HMM->num_states;
        // create vec for beta vals
        beta[q][t].Destroy_And_ReDim(Nq);
        outprob = obs_lprob[q][t+1];

        if (q==startq) beta[q][t][Nq-1]= LOGZERO;
        else beta[q][t][Nq-1]= beta[q+1][t][0];

        Assert(Nq>=2);
        for (i=Nq-2;i!=(t_index)(-1);i--)
        {
            x = act_HMM->trans_mat[i][Nq-1] + beta[q][t][Nq-1];
            if (q>=qLo[t+1] AND q<=qHi[t+1])
                for (j=1;j<Nq-1;j++)
                {
                    a = act_HMM->trans_mat[i][j];
                    y = beta[q][t+1][j];
                    if (a>LOGSMALL AND
y>LOGSMALL)
                        x = LogAdd(x,a
+outprob[j]+y);

                } // endfor j

            beta[q][t][i] = x;
            if (x>lMax) lMax = x;
            if (x>gMax)
            {
                gMax = x;
                q_at_gMax = q;
            }
        } // endfor i
        maxP[q] = lMax;

```

```

        } // endfor q

        last_q = endq;

        while (gMax-maxP[startq] > pruning_threshold)
            startq-=1; // lower startq till threshold
reached
        qHi[t] = startq;
        while ( ((gMax-maxP[endq]) > pruning_threshold) AND endq<t)
            endq+=1; // raise endq till thresh reached
        qLo[t] = endq;

        Compute_Obs_LProbs(whole_file[t], t, qHi[t], qLo[t],
label_idxes_list);

    } // endfor t

    // compute total probability pr
    pr = LOGZERO;
    outprob = obs_lprob[0][0];
    for (j=1;j<Nq-1;j++)
    {
        a = act_HMM->trans_mat[0][j];
        y = beta[last_q][0][j];
        if ( (a>LOGSMALL) AND (y>LOGSMALL) )
            pr = LogAdd(pr,a+outprob[j]+y);
    }

    if (LOGZERO >= pr)
    {
        mwarn<<"Prune threshold = "<<pruning_threshold<<" too
small.";
        return pr;
    }

    return pr;
}

// Setotprob: allocate and calculate otprob matrix at time t
void ModelsSimultaneousTraining::Compute_Obs_LProbs(const VetDouble& obs,
                                                    const t_index t, const t_index beam_top,
                                                    const t_index beam_bottom, const VetULong&
label_idxes_list)
{
    t_signed q;
    t_index j, Nq, endq;

```

```

VetDouble temp_dvet;
EmbCodebook *act_HMM;

if (0==beam_bottom) endq = 0;
else endq = beam_bottom-1;

for (q=beam_top; q>=(t_signed)endq; q--)
{
    act_HMM = &HMM_defs[label_idxes_list[q]];
    Nq = act_HMM->num_states;

    obs_lprob[q][t].Destroy_And_ReDim(Nq-1);
    for (j=1;j<Nq-1;j++)
        obs_lprob[q][t][j]=(*act_HMM)[j-1].Obs_LProb(obs);
}
return ;
}

```

```

//
*****
//
//                                ALPHA MATRIX
//
//
*****

```

```

// InitAlpha: allocate and initialise alfa columns for time t=1
void ModelsSimultaneousTraining::Init_Alpha(const VetULong&
label_idxes_list)

```

```

{
    t_index i,j,q, Nq;

    VetDouble aq,outprob;
    EmbCodebook *act_HMM;
    t_real x;

    // Create Storage Space - two columns
    act_alfa.Destroy_And_ReDim(Q);
    prev_alfa.Destroy_And_ReDim(Q);
    for(q=0;q<Q;q++)
    {
        Nq = HMM_defs[label_idxes_list[q]].num_states;
        act_alfa[q].Destroy_And_ReDim(Nq);
        act_alfa[q].Set(LOGZERO); // initialize act_alfa to

```

LOGZERO

```

        prev_alfa[q].Destroy_And_ReDim(Nq);
    }
    // Calculate prev_alfa (alfat) values for t=0

    act_HMM = &HMM_defs[label_idxes_list[0]];
    Nq = act_HMM->num_states;
    act_alfa[0][0] = LOGZERO;

    if(obs_lprob[0][0].Dim()==0)
        merr<<"No observation prob matrix at time t=0.";

    for (j=1;j<Nq-1;j++)
        if (act_HMM->trans_mat[0][j]>LOGSMALL)
            act_alfa[0][j] = obs_lprob[0][0][j]+act_HMM-
>trans_mat[0][j];
        else act_alfa[0][j] = LOGZERO;

    x = LOGZERO;
    for (i=1;i<Nq-1;i++)
        if (act_HMM->trans_mat[i][Nq-1]>LOGSMALL)
            x = LogAdd(x, act_alfa[0][i] + act_HMM->trans_mat
[i][Nq-1]);

    act_alfa[1][0] = x;
    Zero_Alpha(1,Q-1, label_idxes_list);

    return;
}

```

```

// StepAlpha: calculate alfat column for time t and return
// forward beam limits in startq and endq
// for first opbsevation startq = endq=0
// startq and endq must be passed by reference to update their value
void ModelsSimultaneousTraining::Compute_Alpha_At_Time_t(t_index& startq,
t_index& endq,

```

```

const t_real pr, const
VetULong& qHi, const VetULong& qLo,
const t_index t, const
VetULong& label_idxes_list)
{
    Bi_D_List tmp;
    EmbCodebook *act_HMM;
    t_index i,j,q,Nq;
    t_real a,x,y;
    startq = qLo[t-1];

```

```

while (pr - MaxModelProb(startq, t-1, label_idxes_list)>MINMODELPROB)
    startq+=1;
if (startq<qLo[t])
    startq = qLo[t];
endq = qHi[t-1];

while (pr - MaxModelProb(endq, t-1, label_idxes_list)>MINMODELPROB)
    endq-=1;
if (endq>qHi[t])
    endq = qHi[t];

tmp = prev_alfa; prev_alfa = act_alfa; act_alfa = tmp;
act_alfa[0][0] = LOGZERO;

if (startq>0) Zero_Alpha(0, startq-1, label_idxes_list);

// at any time the t-th column of alpha matrix is calculated only
for that
// model states in the range [startq; endq]; out of that range the
alfa
// values are not meaningful

for (q = startq; q <= endq; q++)
{
    act_HMM = &HMM_defs[label_idxes_list[q]];
    Nq = act_HMM->num_states;
    if (obs_lprob[q][t].Dim() == 0)
        merr<<"Bug outprob NULL at time t in StepAlpha.";

    for (j=1;j<Nq-1;j++)
    {
        x = LOGZERO;
        for (i=0;i<Nq-1;i++)
        {
            a=act_HMM->trans_mat[i][j];
            y = prev_alfa[q][i];
            if(a>LOGSMALL AND y>LOGSMALL)
                x = LogAdd(x, y + a);
        }
        act_alfa[q][j] = x + obs_lprob[q][t][j];
    }
    x = LOGZERO;
    for (i=1;i<Nq-1;i++)
    {

```

```

        a=(act_HMM->trans_mat[i][Nq-1]);
        y = act_alfa[q][i];
        if(a>LOGSMALL AND y>LOGSMALL)
            x = LogAdd(x,y+a);
    }
    act_alfa[q][Nq-1] = x;
    if (q<Q-1)
        act_alfa[q+1][0] = x;
    }
    if (endq<Q-1) Zero_Alpha(endq+1,Q-1, label_idxes_list);

    return;
}

```

// ZeroAlpha: zero alpha's of given models

```

void ModelsSimultaneousTraining::Zero_Alpha(const t_index qlo, const
t_index qhi,

```

```

                                const VetULong& label_idxes_list)
{
    t_index j, q, Nq;

    // qlo model - dont erase carry over
    Nq = HMM_defs[label_idxes_list[qlo]].num_states;

    for (j=1;j<Nq;j++)
        act_alfa[qlo][j] = LOGZERO;
    for (q=qlo+1;q<=qhi;q++)
    {
        // general case - all states LZERO
        Nq = HMM_defs[label_idxes_list[q]].num_states;

        for (j=0;j<Nq;j++)
            act_alfa[q][j] = LOGZERO;
    }
    if (qhi<Q-1)
        act_alfa[qhi+1][0] = LOGZERO;
    return;
}

```

// ***** END ALPHA MATRIX *****

// GetOutVec: Get observation vector obs

```

void ModelsSimultaneousTraining::Get_Observation(VetDouble& obs,
                                const Bi_D_List& whole_file, const t_index

```

t)

```
{
  obs = whole_file[t];
  return;
}
```

```
// MaxModelProb: Calc max probability of being in model q at
// time t, return LOGZERO if cannot do so
t_real ModelsSimultaneousTraining::MaxModelProb(const t_index q, const
t_index t,
```

```
const VetULong& label_idxes_list)
```

```
{
  t_real maxP,x;
  t_index Nq,i;

  maxP = LOGZERO;

  if(q<beta.Dim())
  {
    Nq = HMM_defs[label_idxes_list[q]].num_states;

    if (beta[q][t].Dim()!=0)
    {
      for (i=0;i<Nq-1;i++)
        if ((x=act_alfa[q][i]+beta[q][t][i]) >
maxP)
          maxP = x;
    }
  }

  return maxP;
}
```

```
// *****
//
//          FIRST (HIGHER) LEVEL FUNCTION
//          Update_Models
//
// *****
```

```
void ModelsSimultaneousTraining::Update_Models()
{
  t_index h, n;
```

```

// for each dbase label-> i.e. for each file
for (h=0;h<HMM_defs.Dim();h++)
    {
        n = HMM_accs[h].num_instances;

        if (n < min_istance_number )
            {
                mwarn<<"Insufficient training examples for model.";
                mwarn<<"Model "<<h<<" copied: only "<<n<<" examples";
            }
        else{
            Update_Model(HMM_defs[h], HMM_accs[h]);
            mstat<<"Model "<<h<<" updated with "<<n<<" examples
\n";
        }
    }

return;
}

```

```

// *****
//
//                      SECOND LEVEL FUNCTIONS
//                      Update_Models
//
// *****

```

```

void ModelsSimultaneousTraining::Up_Mix_Parms(EmbCodebook& hmm,
StatisticsAccumulators& acc,

```

```

    const t_index q, const t_index t,

    const VetDouble& obs, const t_real pr)
    {
        t_index i,j,k,kk,m,N, obs_size;
        VetDouble aqt1;
        t_real log_gamma, initx;
        t_real gamma, w, zmean,zmean1,zmean2;
        VetDouble mean;

        N = hmm.num_states;

        if(t==0) aqt1.Reset();
        else aqt1=prev_alfa[q];
    }

```

```

    for (j=1;j<N-1;j++)
    {
        if ((t==0) AND (q==0))
            initx = hmm.trans_mat[0][j];
        else if ((t==0) AND (q!=0)) //
            shouldn't happen
            initx = LOGZERO;
        else{
            initx = LOGZERO;
            for (i=0;i<N-1;i++)
                initx = LogAdd(initx, aqt1[i] +
hmm.trans_mat[i][j]);
        }
        if (initx>LOGSMALL)
            for (m=0;m<hmm[j-1].Dim();m++)
            {
                w = hmm[j-1][m].weight;
                if (w>LMINMIX)
                {
                    mean = hmm[j-1][m].mean;
                    log_gamma = initx + w + beta[q][t]
[j];

                    if (log_gamma>LOGSMALL)
                    {
                        log_gamma += hmm
[j-1].Mix_Obs_LProb(obs,m)-pr;

                        if (log_gamma>MINEXPARG)
                        {
                            gamma = exp

                                obs_size = acc.mu

                                for

                                    {
                                        if

                                            acc.mu

                                [j][m][k] += obs[k]*gamma;

                                    if

                                (reest_variances)

                                    {
                                        if

```

```

(!hmm.full_covariance)
{
    zmean=obs[k]-mean[k];

    acc.diag_va[j][m][k] += (zmean*zmean*gamma);
}else {

    zmean1 = obs[k]-mean[k];

    for (kk=k;kk<obs_size;kk++)

        {

            zmean2 = obs[kk]-mean[kk];

            acc.full_cov[j][m][k][kk]+= (zmean1*zmean2*gamma);

        } // endfor kk

    } // endif (!hmm.Full_Cov())

    // endif (reest_variances)

    } // endfor k
    if (reest_weights)
        acc.c[j][m]
+=gamma;
    } // if
(log_gamma>MINEXPARG)
    } // endif (x>LOGSMALL)
    }
    } // endfor m
    } // endfor j
    return;
}

```

```

// UpTranParms: update the transition counters of given acc
void ModelsSimultaneousTraining::Up_Tran_Parms(const EmbCodebook& hmm,
StatisticsAccumulators& acc,

```

```

    const t_real pr, const t_index q, const t_index Q,
    const t_index t, const t_index T)

```

```

    {
    t_index i,j,N;
    t_real gamma;
    VetDouble bqt1;

    N = hmm.num_states;

    if(t==T-1)
    {
        bqt1.Reset();
    }
    else bqt1 = beta[q][t+1];

    if (t<T-1)
    {
        for (i=0;i<N-1;i++)
        {
            for (j=1;j<N-1;j++)
            {
                if (i==0 AND q==0)
                {
                    if (t==0)
                    {
                        gamma = act_alfa[q][j]+beta
[q][t][j]-pr;
                        if (gamma>MINEXPARG)
                        acc.tran[i][j] += exp(gamma);
                    }
                }
            }
        }
        else{
            if (bqt1.Dim() != 0)
            {
                gamma = act_alfa[q][i]
+hmm.trans_mat[i][j]+obs_lprob[q][t+1][j]+bqt1[j]-pr;
                if (gamma>MINEXPARG)
                acc.tran[i][j] += exp(gamma);
            }
        }
        // endif (i==0 AND q==0)
    } // endfor j
    if (q<Q-1)
    {
        gamma = act_alfa[q][i]+ hmm.trans_mat[i]
[N-1]+beta[q][t][N-1]-pr;
        if (gamma>MINEXPARG) acc.tran[i][N-1] +=
exp(gamma);
    }
}

```

```

        }
    } // endfor i
}
else{
    if (q==Q-1)
        for (i=1;i<N-1;i++)
            {
                gamma = act_alfa[q][i]+beta[q][t][i]-pr;
                if (gamma>MINEXPARG) acc.tran[i][N-1] +=
exp(gamma);
            }
}
return;
}

```

// UpOccCount: update the occupation counters of given acc
void ModelsSimultaneousTraining::Update_Occourrence_Counter
(StatisticsAccumulators& acc,

```

const t_index t, const t_index q, const t_real pr)
{
    t_index i,N;
    t_real gamma;

    N = acc.occ.Dim();
    if (q==0)
    {
        if (t==0) acc.occ[0]+=1;
    }
    else{
        gamma = act_alfa[q][0]+beta[q][t][0]-pr;
        if (gamma>MINEXPARG) acc.occ[0] += exp(gamma);
        for (i=1;i<N-1;i++)
            {
                gamma = act_alfa[q][i]+beta[q][t][i]-pr;
                if (gamma>MINEXPARG) acc.occ[i] += exp(gamma);
            }
    }

    return;
}

```

// ----- Model Update -----

```

void ModelsSimultaneousTraining::Update_Model(EmbCodebook& hmm,
StatisticsAccumulators& acc)
{
    t_index i, j, k, kk, m, N, obs_size;
    t_real new_aij, new_mix_weight, new_var, occi,c_im,cfloor;

    VetDouble mean;
    MatrixOfDouble covariance;
    VetDouble mu_im,va_im_var,va_im_inv_k;
    MatrixOfDouble va_im_inv;

    cfloor = MINMIX;
    N = hmm.num_states;

    if (reest_transitions)
    {
        hmm.trans_mat.Set(LOGZERO);
        for (i=0;i<N-1;i++)
        {
            occi = acc.occ[i];
            if (occi > 0.0)
                for (j=1;j<N;j++)
                {
                    new_aij = acc.tran[i][j]/occi;
                    if(new_aij>MINLOGARG)
                        hmm.trans_mat[i][j] = log
(new_aij);

                    else hmm.trans_mat[i][j] = LOGZERO;
                }
            else mwarn<<"Model "<<hmm.file<<" state "<<i<<"
never occupied.";
        } // endfor i
    } // endif (reest_transitions)

    if (reest_means OR reest_variances OR reest_weights)
    {
        // for each effective spectral state
        for (i=1;i<N-1;i++)
        {
            occi=acc.occ[i];
            if (occi > 0.0)
            {
                M= hmm[i-1].Dim();
                // for each gaussian of i-th state
                for (m=0;m<M;m++)
                {

```

```

if (M==1)
    c_im = occi;
else c_im = acc.c[i][m];
new_mix_weight = c_im/occi;
if (reest_weights)
{
    if (new_mix_weight > 1.0)
    {
        if (new_mix_weight
> 1.001)
// this is
serious!
mwarn<<"Mix
too big (new_mix_weight = "<<new_mix_weight
<<") in model n. "<<hmm.file<<" state "<<i
<<", mix "<<m;
new_mix_weight = 1.0;
}

if
    hmm[i-1][m].weight
= LOGZERO;
else hmm[i-1][m].weight =
log(new_mix_weight);
}

if (new_mix_weight >= cfloor)
{
    if (reest_means)
        hmm[i-1][m].mean =
acc.mu[i][m]/(t_real)c_im;

    if (reest_variances)
    {
        if (!
hmm.full_covariance)
{
obs_size =
hmm[i-1][m].diag_inv_cov.Dim();
for
(k=0;k<obs_size;k++)
{

```

```

new_var = acc.diag_va[i][m][k]/c_im;
                                                                    if
(new_var > min_var_value)

    hmm[i-1][m].diag_inv_cov[k][k]= new_var;

else hmm[i-1][m].diag_inv_cov[k][k]=  const_to_add_min_var+new_var;
                                                                    }

    // endfork
                                                                    }
                                                                    else{
                                                                    obs_size =
                                                                    for
                                                                    for
                                                                    {
                                                                    new_var = acc.full_cov[i][m][k][kk]/c_im;
                                                                    if(k == kk AND new_var < min_var_value)
                                                                    new_var = min_var_value;
                                                                    hmm[i-1][m].inv_cov[k][kk]
                                                                    = hmm[i-1][m].inv_cov[kk][k] = new_var;
                                                                    }
                                                                    }
                                                                    if(!hmm[i-1]
[m].Compute_G_Const())
mwarn<<"Invalid inverse matrix in file: "<<hmm.file
                                                                    <<"
state: "<<i<<" gauss: "<<m;
                                                                    } // endif
(reest_variances)
                                                                    } // endif (new_mix_weight
>= cfloor)
                                                                    } // endfor m
                                                                    } // endif (occi>0)
else mwarn<<"Model "<< hmm.file <<" state" << i<< "

```

```

never occupied. \n";
        } // end for i
    } //endif (reest_means OR reest_variances OR
reest_mixtures)
    return;
}

```

```

// ***** UPDATE MODELS *****

```

```

void ModelsSimultaneousTraining::Store_Statistic_Accs(const String&
accs_file)
{
    t_index i,j,h,k,z,Nh,Mh;
    t_index obs_size;
    ofstream file;

    file.open(accs_file);
    file.precision(OUTPUT_SIZE);

    for(h=0;h<HMM_accs.Dim();h++)
    {
        Nh=HMM_accs[h].num_states;
        Mh=HMM_accs[h].num_mixes;

        file<<"file: "<<h<<"\n\n";
        file<<"num_instances= "<<HMM_accs[h].num_instances<<"\n";
        file<<"num_states= "<<Nh<<"\n";
        file<<"num_mixes= "<<Mh<<"\n\n";
        file<<"tran:\n";

        for(i=0;i<Nh-1;i++)
        {
            for(j=1;j<Nh;j++)
                file<<HMM_accs[h].tran[i][j]<<" ";
            file<<"\n";
        }

        file<<"\nocc: ";
        for(i=0;i<Nh-1;i++)
            file<<HMM_accs[h].occ[i]<<" ";

        obs_size=HMM_accs[h].mu[0][0].Dim();
    }
}

```

```

file<<"\n\nmu:\n";
for(i=1;i<Nh-1;i++)
    for(j=0;j<Mh;j++)
        {
            for(k=0;k<obs_size;k++)
                file<<HMM_accs[h].mu[i][j][k]<<" ";
            file<<"\n";
        }

if(HMM_accs[h].full_cov.Dim() != 0)
    {
        file<<"\nfull_cov: \n";
        for(i=1;i<Nh-1;i++)
            for(j=0;j<Mh;j++)
                for(k=0;k<obs_size;k++)
                    {
                        for(z=k;z<obs_size;z++)
                            file<<HMM_accs
[h].full_cov[i][j][k][z]<<" ";
                        file<<"\n";
                    }
    }
else{
    file<<"\n\ndiag_va: \n";
    for(i=1;i<Nh-1;i++)
        for(j=0;j<Mh;j++)
            {
                for(k=0;k<obs_size;k++)
                    file<<HMM_accs[h].diag_va
[i][j][k]<<" ";
                file<<"\n";
            }
}

file<<"\nc: \n";
for(i=1;i<Nh-1;i++)
    {
        for(j=0;j<Mh;j++)
            file<<HMM_accs[h].c[i][j]<<" ";
        file<<"\n";
    }
file<<"\n";
}

file.close();
return;

```

```

    }

void ModelsSimultaneousTraining::Load_Models_Parameters()
{
    t_index symbol, num_symbols;
    t_index vec_size;
    String buffer;
    Boolean use_full_cov;
    ifstream init_spcf;

    init_spcf.open(models_file_input, ios::in|ios::nocreate);
    Read_Data_File_Header (init_spcf, vec_size, use_full_cov);

    if(features.Feature_Vet_Dim()!=vec_size)
        merr<<"Not compatible statistics dimension with initialized
acoustic models";

    Write_Header_Of_File_Model(models_file_output, dbase.Snd_Type(),
        dbase.Label_Type(), dbase.Db_File_List_Name(),
dbase.Window_Lenght(),
        dbase.Window_Overlap(), vec_size, use_full_cov);

    num_symbols = HMM_defs.Dim();
    HMM_accs.Destroy_And_ReDim(num_symbols);

    for(symbol=0; symbol<num_symbols; symbol++)
    {
        HMM_defs[symbol].file=symbol;
        HMM_defs[symbol].stat_dim=vec_size;
        HMM_defs[symbol].full_covariance=use_full_cov;
        HMM_defs[symbol].Read(init_spcf, use_full_cov);
        HMM_accs[symbol].Configure(HMM_defs
[symbol].num_states,HMM_defs[symbol].num_gauss,
            vec_size, use_full_cov);
    }

    return;
}

```

```

void ModelsSimultaneousTraining::Load_Statistic_Accs(const String&
accs_file)
{
    t_index i,j,h,k,z,Nh,Mh;
    t_index obs_size, file;

```

```

    ifstream file;
String buffer;
    t_real val;

    file.open(accs_file,ios::in|ios::nocreate);
    if(file.fail())
        merr<<"Could not open file of statistics accumulators.";

    file.precision(OUTPUT_SIZE);

    for(h=0;h<HMM_accs.Dim();h++)
    {
        file>>buffer;
        file>>file;
        Assert(file==h);

        file>>buffer;
        file>>HMM_accs[h].num_instances;
        file>>buffer;
        file>>Nh;
        file>>buffer;
        file>>Mh;
        file>>buffer;
        for(i=0;i<Nh-1;i++)
            for(j=1;j<Nh;j++)
            {
                file>>val;
                HMM_accs[h].tran[i][j]+=val;
            }

        file>>buffer;
        for(i=0;i<Nh-1;i++)
        {
            file>>val;
            HMM_accs[h].occ[i]+=val;
        }

        obs_size=HMM_accs[h].mu[0][0].Dim();
        file>>buffer;
        for(i=1;i<Nh-1;i++)
            for(j=0;j<Mh;j++)
                for(k=0;k<obs_size;k++)
                {
                    file>>val;
                    HMM_accs[h].mu[i][j][k]+=val;
                }
    }

```

```

        file>>buffer;
        if(buffer=="full_cov:")
            for(i=1;i<Nh-1;i++)
                for(j=0;j<Mh;j++)
                    for(k=0;k<obs_size;k++)
                        for(z=k;z<obs_size;z++)
                            {
                                file>>val;
                                HMM_accs
[h].full_cov[i][j][k][z]+=val;
                            }
            else for(i=1;i<Nh-1;i++)
                for(j=0;j<Mh;j++)
                    for(k=0;k<obs_size;k++)
                        {
                            file>>val;
                            HMM_accs[h].diag_va[i][j][k]
+=val;
                        }

```

```

        file>>buffer;
        for(i=1;i<Nh-1;i++)
            for(j=0;j<Mh;j++)
            {
                file>>val;
                HMM_accs[h].c[i][j]+=val;
            }
        }

        file.close();
        return;
    }

```

```

// *****
//
//          CLASS EmbNodeSpecShape
//
//
// *****

```

```

// MOutP: Returns prob of vector x for given state & given mixture
t_real EmbNodeSpecShape::Mix_Obs_LProb(const VetDouble& obs, const t_index
m)

```

```

{
t_real prob;

prob=(*this)[m].Evaluate_Exp_Gauss(obs);
prob+=(*this)[m].gConst;

return prob;
}

```

// OutP: Returns probability (log) of vector x for given state

```

t_real EmbNodeSpecShape::Obs_LProb(const VetDouble& obs)
{
t_index m;
t_real bx,px;

bx = LOGZERO;                                     /* Multi Mixture Case */
for (m=0; m<Dim(); m++)
{
    if ((*this)[m].weight>LMINMIX)
    {
        px =(*this)[m].Evaluate_Exp_Gauss(obs);
        px += (*this)[m].gConst;
        bx = LogAdd(bx, (*this)[m].weight + px);
    }
}

return bx;
}

```